Software Production Engineering
Course Project

IMT2020110 Vivek Tagudu

IMT2020085 Harshadeep Donapati

Introduction

This is a Library Management System designed in React for adding books, issuing books, editing issues, and more. We used the MERN stack (MongoDB, Express, React, and NodeJS) to build this application.

Github Repo Link: https://github.com/harsha-deep/Lib

Deployed Application Link: https://library-odfb.onrender.com/

DevOps tools:

Source Control Management: Git and GitHub

Continuous Integration Pipeline: Jenkins

Containerization: Docker

Container Orchestration: Docker compose

Front End: React, Tailwind CSS, Antd

Logger: Winston

Monitoring: ELK Stack

Database: MongoDB

Testing: Chai, Jest

Other dependencies

- **bcryptjs -** For password hashing
- **jsonwebtoken** - For JWT auth
- **express -** For the backend server
- **mongoose** - For MongoDB operations
- **dotenv** - Dealing with environment variables

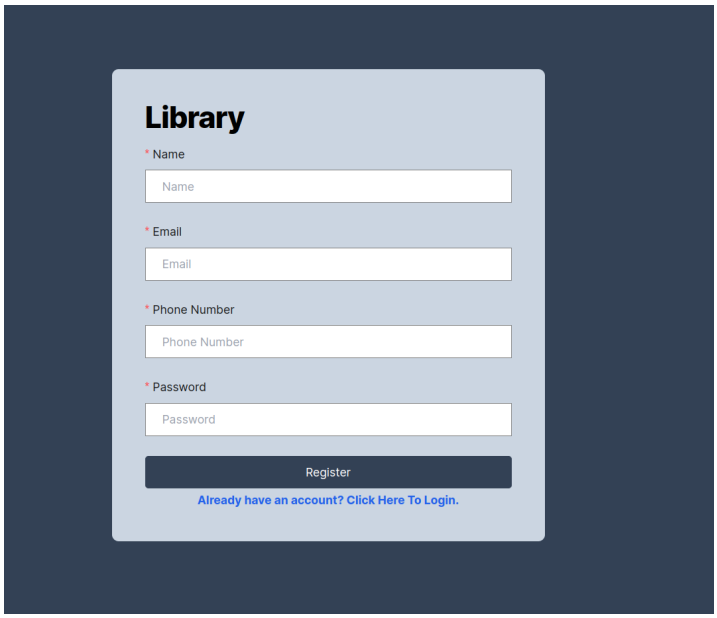
Features

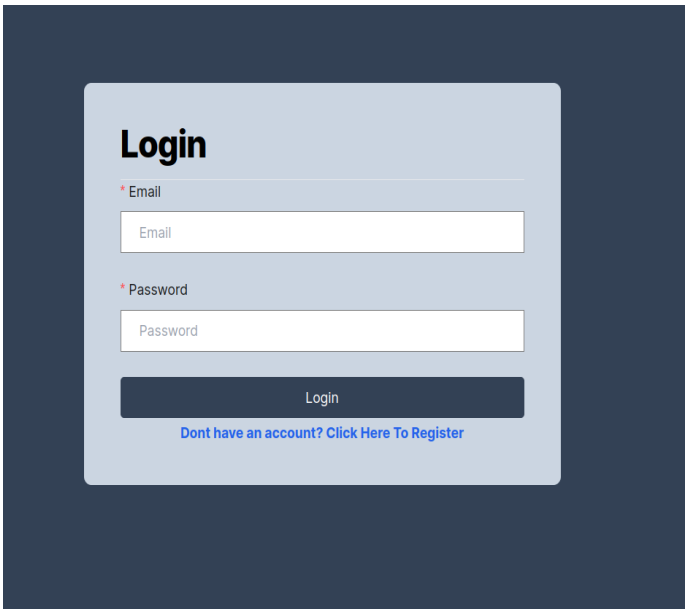
- **Register**

  New Users can register with the system.

- **Login**

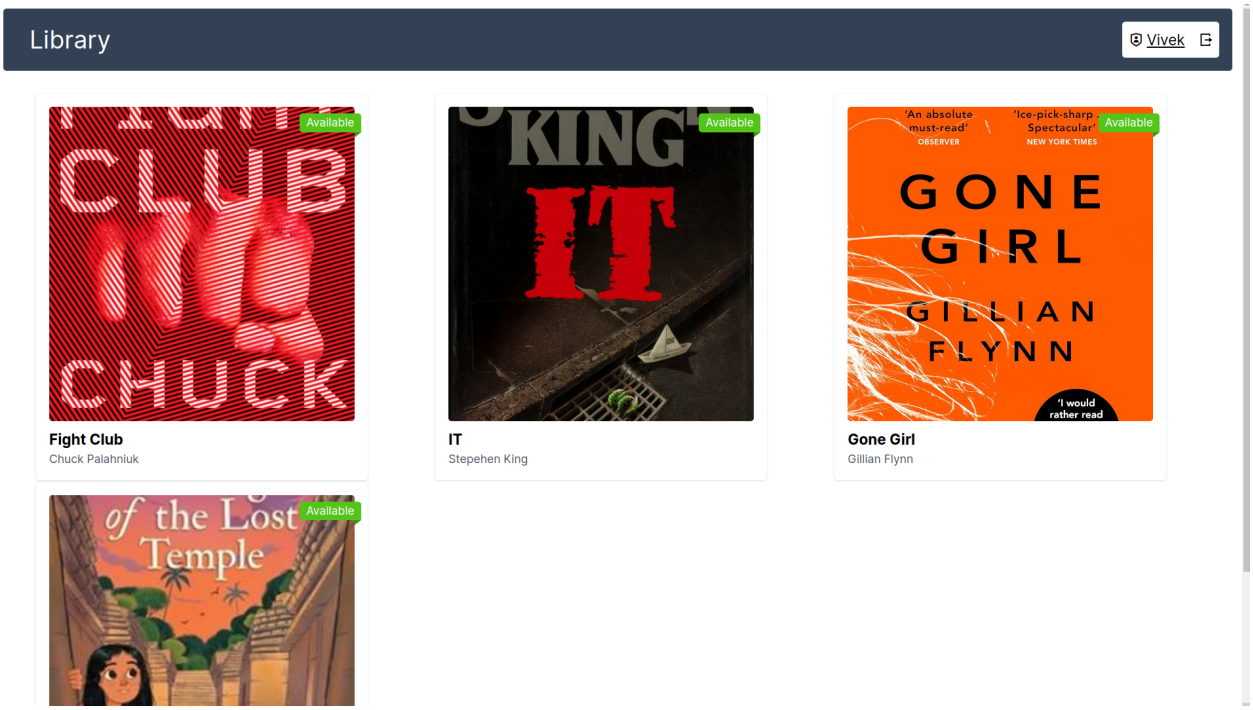
  Registered users can log in to the system with their credentials.





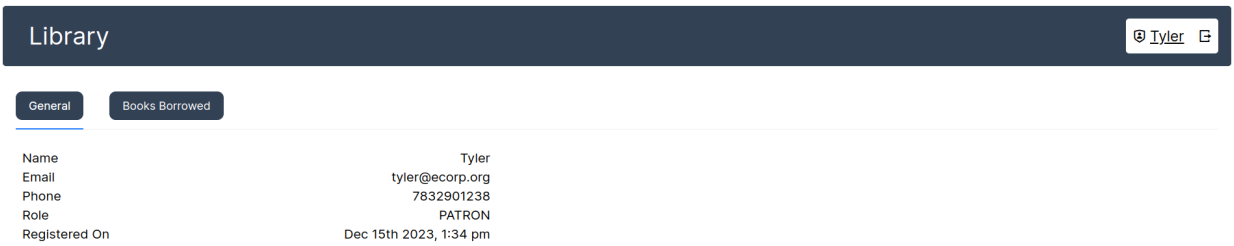
- **List of available books**

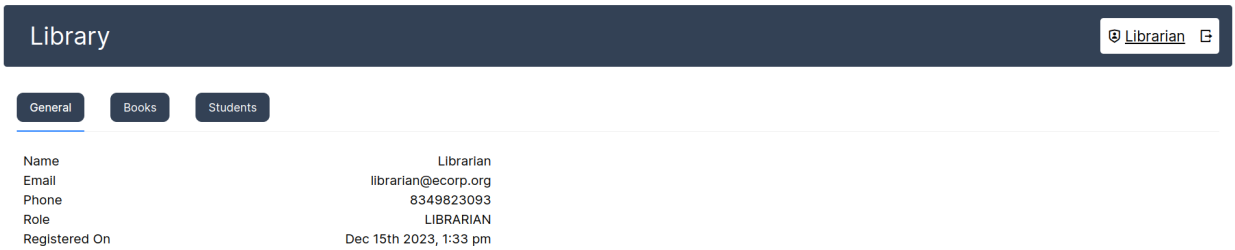
Upon logging in, the user will be displayed a list of books and each book has a tag about their availability. (Available/Not Available)

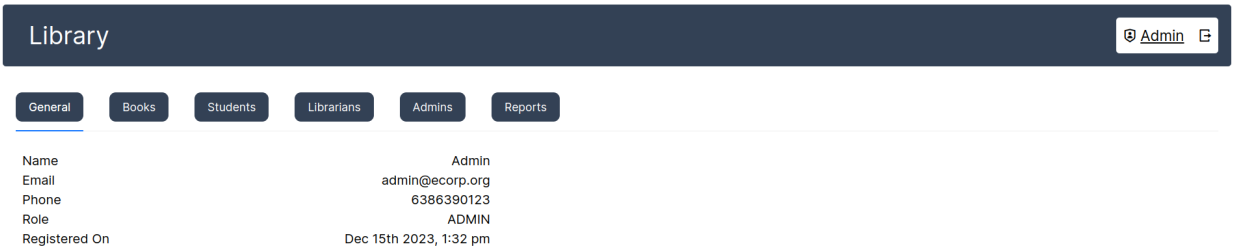


- **Access Management**

  There are 3 types of users - normal users(patron), the librarian, and the admin. All tabs are visible to the admin only. (He has the highest control over the system.)



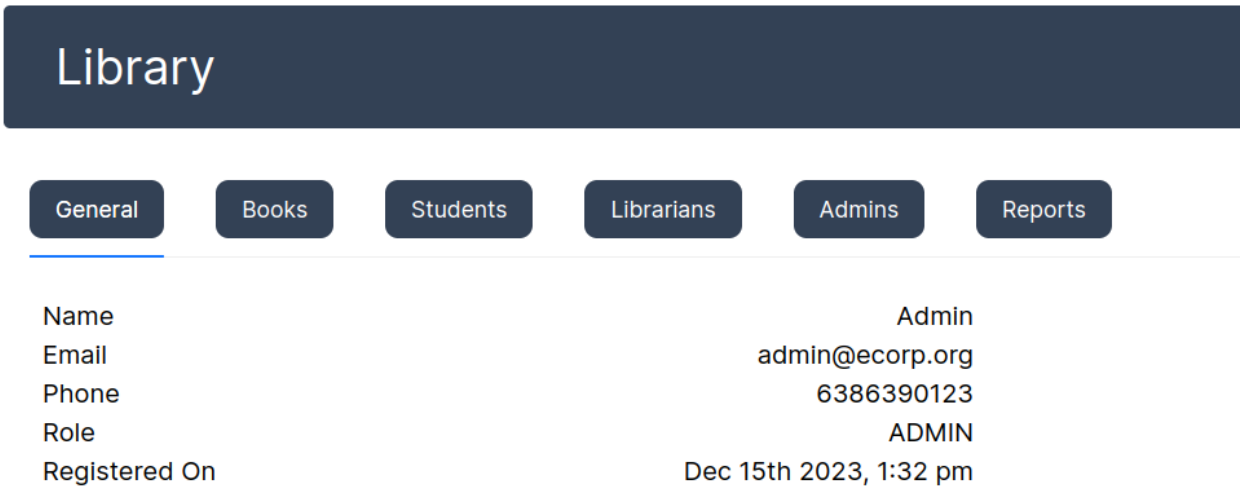




List of tabs

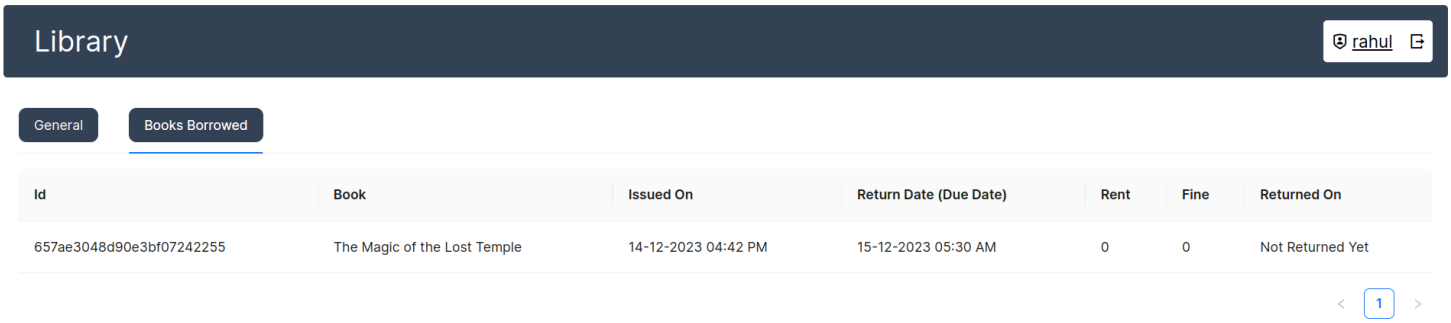
- **General**

  This tab shows the general details of the logged-in user.



- **Books Borrowed tab**

  This tab shows the list of borrowed books by the user.



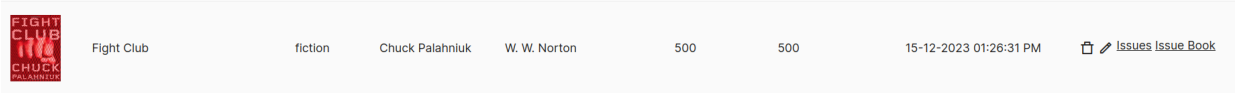
- **Books tab**

  This tab shows the general details of a book (book image, book title, category, author, publisher, total copies, available copies, added date, and various actions (discussed later).

  This tab is not visible to the user.

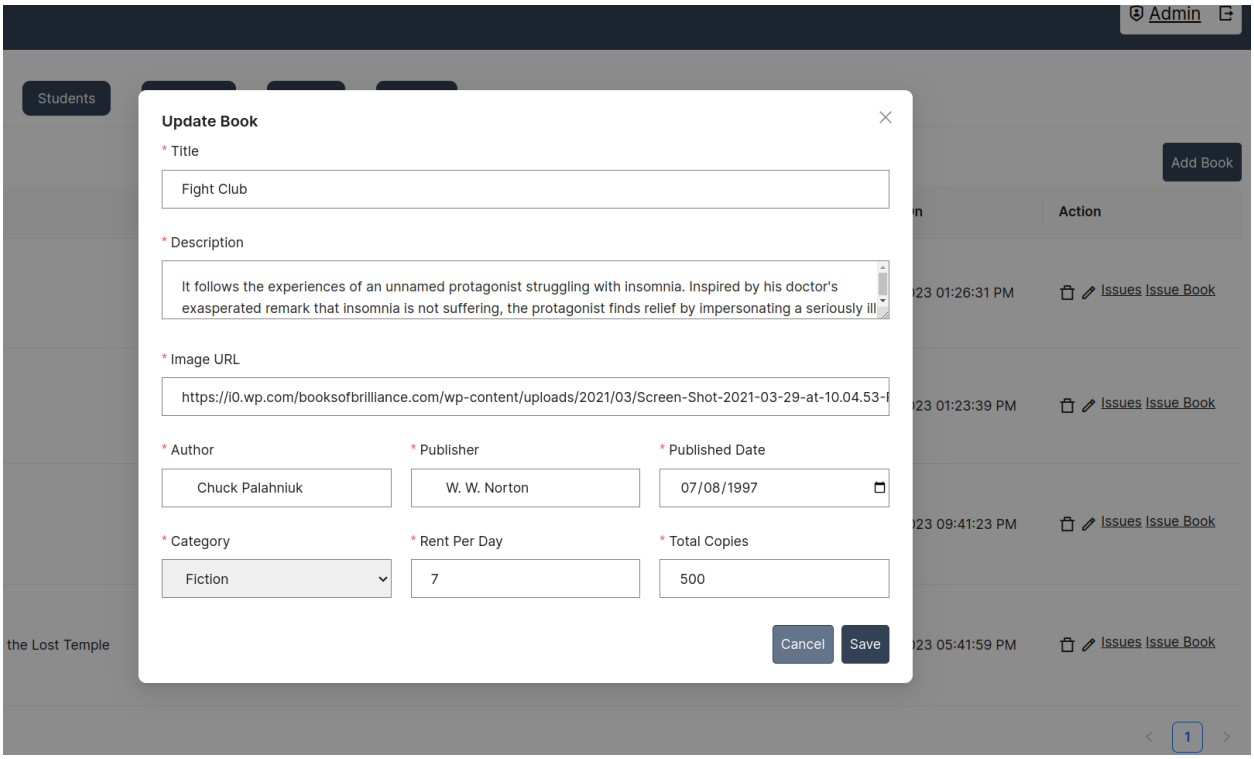
- **Delete book**

  The bin icon (on the right-hand side) can be used to delete the book.



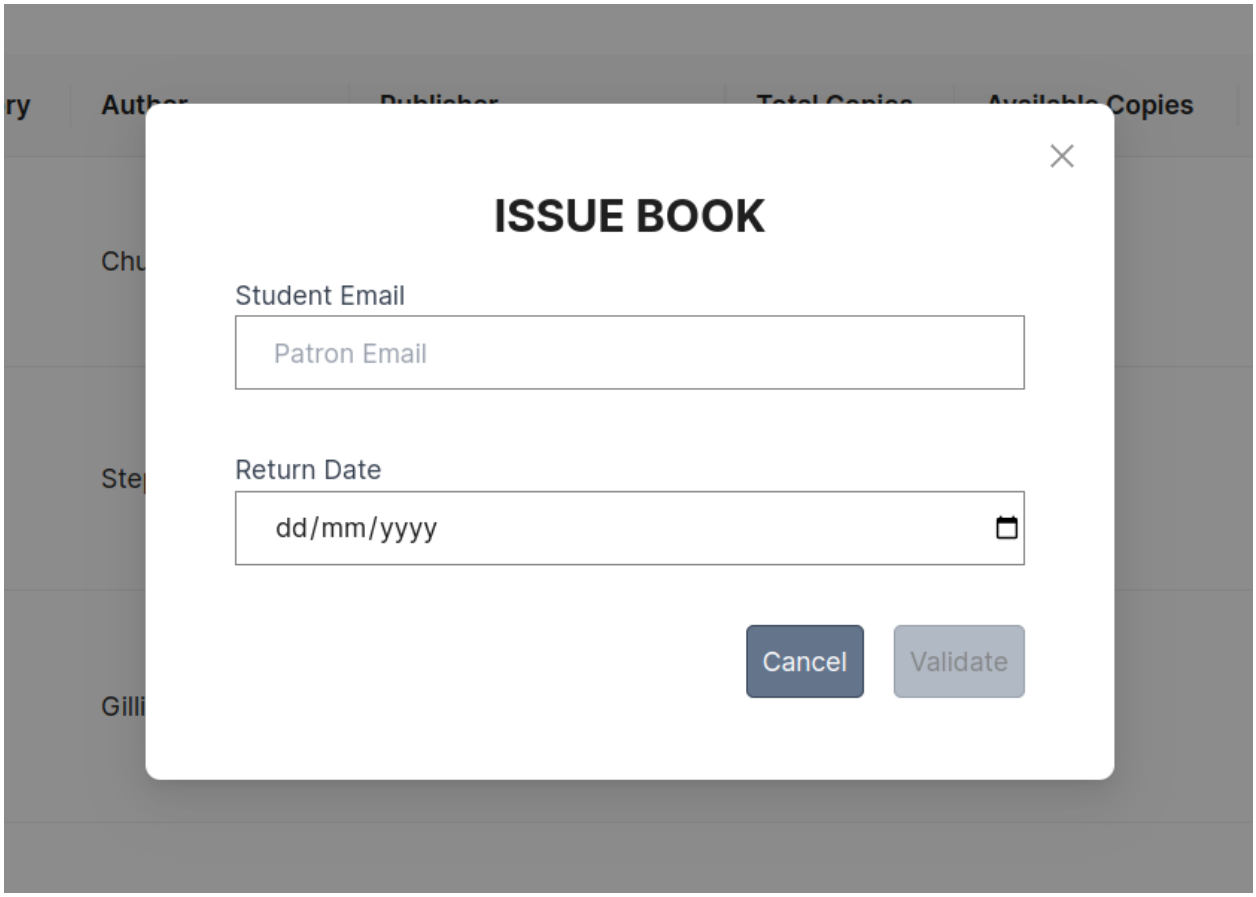
- **Edit book**

  The pencil icon can be used to edit the book details. It renders a popup form with already filled details which can be edited and saved.



- **Issue a book**

  A book can be issued by clicking the issue book link.



- **List of Issues**

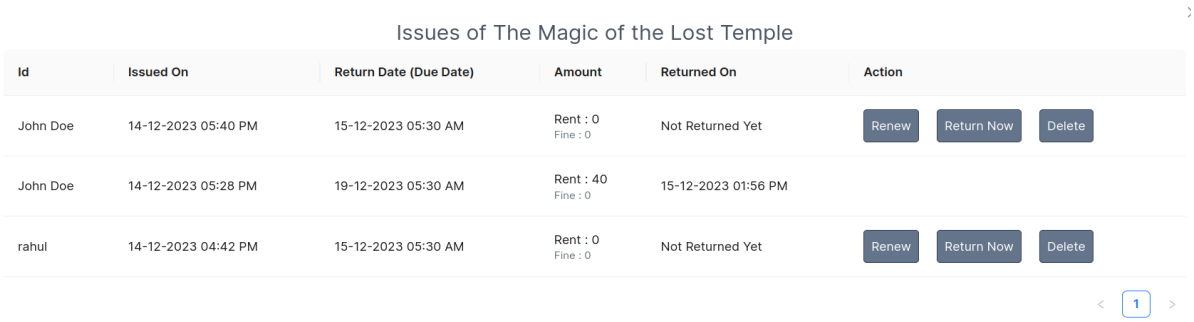
  The admin/librarian can see the issues of the book with some basic details (to whom it was issued, when it was issued etc.)

- **Return the book**

  The admin/librarian can click the return book button if the user has returned a book.

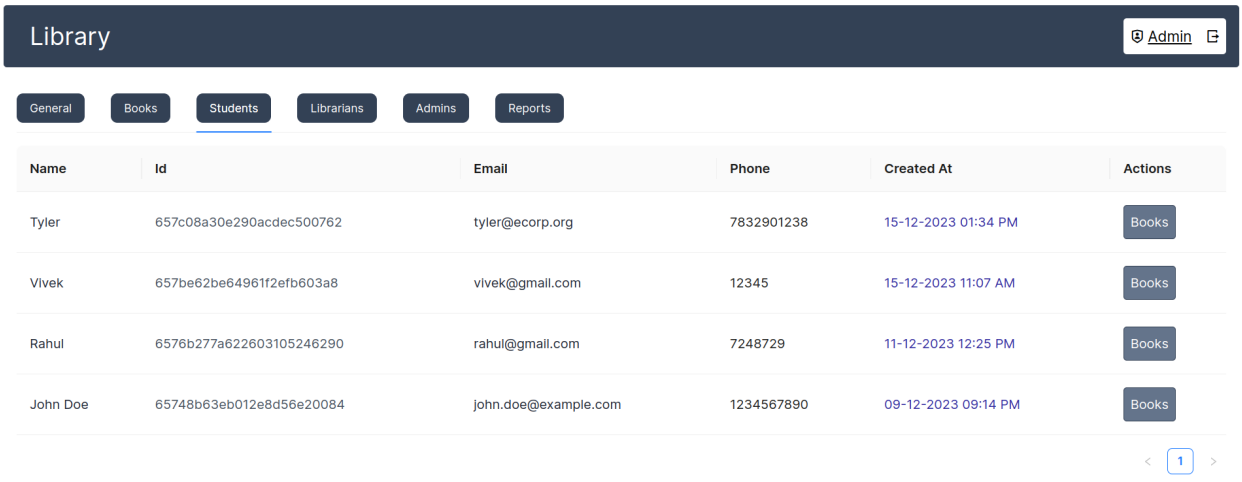
- **Delete an issue**

  The admin/librarian can delete the issue using the delete button.

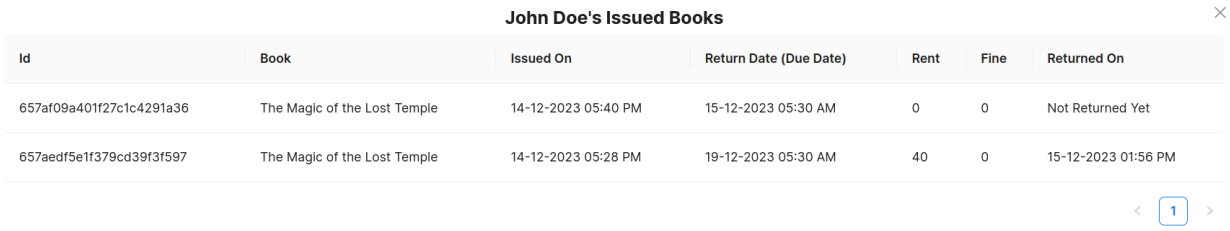


- **Students/Librarians/Admins tab**

  The admin/librarian can see a list of all students with basic details (name, email, phone number, etc.)



  The admin/librarian can also see a user's issued books by clicking the "Books" button.



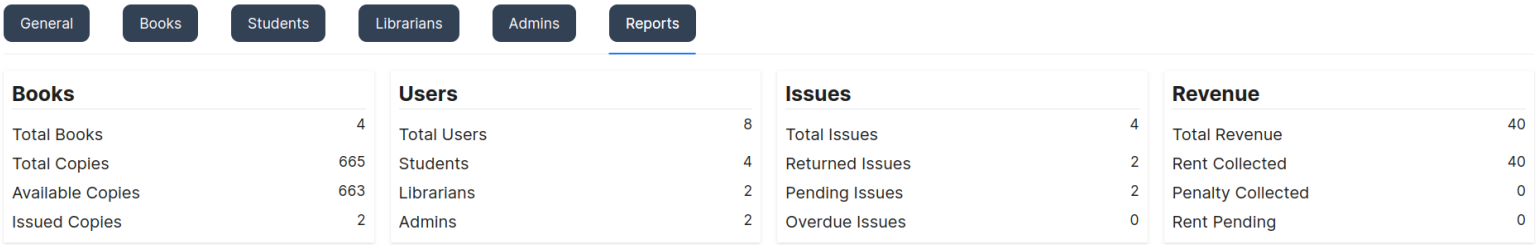
John Doe's Issued Books

Id	Book	Issued On	Return Date (Due Date)	Rent	Fine	Returned On
657af09a401f27c1c4291a36	The Magic of the Lost Temple	14-12-2023 05:40 PM	15-12-2023 05:30 AM	0	0	Not Returned Yet
657aedf5e1f379cd39f3f597	The Magic of the Lost Temple	14-12-2023 05:28 PM	19-12-2023 05:30 AM	40	0	15-12-2023 01:56 PM

1

- **Reports tab**

  Only the admin can see the report tab which contains all the details about the whole system. This includes every detail about books, users, issues, and the whole revenue generated by the system.



General | Books | Students | Librarians | Admins | Reports

Books		Users		Issues		Revenue	
Total Books	4	Total Users	8	Total Issues	4	Total Revenue	40
Total Copies	665	Students	4	Returned Issues	2	Rent Collected	40
Available Copies	663	Librarians	2	Pending Issues	2	Penalty Collected	0
Issued Copies	2	Admins	2	Overdue Issues	0	Rent Pending	0

Docker

Front-end docker file

```
FROM node:21

WORKDIR /app

COPY ./package.json /app
COPY ./package-lock.json /app

RUN npm ci
COPY . /app

EXPOSE 3000
CMD [ "npm", "start" ]
```

Backend docker file

```
FROM node:21

WORKDIR /app
RUN npm install -g nodemon

COPY ./package.json /app
COPY ./package-lock.json /app

RUN npm ci
COPY . /app
EXPOSE 5002

CMD ["nodemon", "server"]

```

This Dockerfile is for a Node.js application. It starts with the Node.js 21 as the base image, sets the working directory to /app, installs the nodemon tool globally for automatic server restarts, and then copies the package.json and package-lock.json files to the container before running npm ci to install dependencies. The application code is then copied into the container, and it exposes port 5002/3000. The final command in the Dockerfile utilizes nodemon to run the backend server and npm to run the frontend server.

Jenkins

We used Jenkins pipeline scm from GitHub. The pipeline script was cloned from the GitHub repository And the code was also cloned from the same repository.

The first step was to clone the repo.

```
pipeline {
    agent any
    environment {
        mongo_url = "mongodb+srv://vivektangudu:viv@cluster0.czt49fi.mongodb.net/"
        JWT_SECRET = "SECRETLEL"
    }
    stages {
        stage('Stage 1: Git Clone') {
            steps {
                git branch: 'main',
                url: 'https://github.com/harsha-deep/Lib'
            }
        }
```

The second step is to build the front-end image. All the contents are required to build the front-end image

```
        stage('client build') {
            steps {
                dir('client'){
                sh "npm install"
                sh 'docker build -t frontend-image .'
            }
            }
        }
```

Similarly, the next step is to build the back-end image.

```
        stage("Server build") {
            steps {
                dir('server'){
                sh "npm install"
                sh 'docker build -t backend-image .'
            }}
        }
```

The next step is to push the docker images to the DockerHub.

```
    stage('Push to Docker Hub') {
        steps {
            script {
                    sh "docker login --username bean6792 --password Vivek@1383"
                    sh 'docker tag frontend-image bean6792/frontend-image:latest'
                    sh 'docker push bean6792/frontend-image:latest'
                    sh "docker tag backend-image bean6792/backend-image:latest"
                    sh "docker push bean6792/backend-image:latest"

            }
        }
    }
```

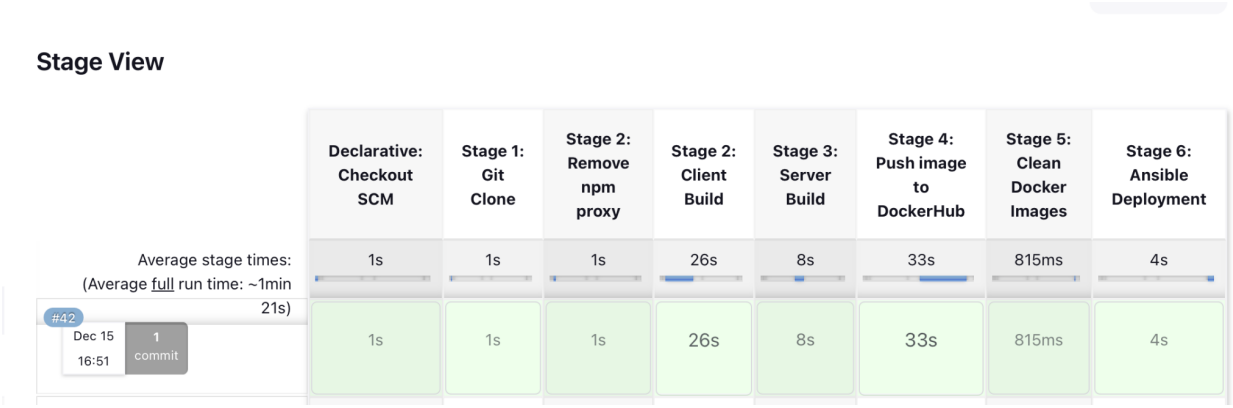
```
stage('Clean Docker Images') {
    steps {
        script {
            sh 'docker container prune -f'
            sh 'docker image prune -f'
        }
    }
}
```

We can use the docker [image/container] prune command to delete old docker and dangling docker images/containers.

Using Ansible, we can start the docker images by assigning the port and declaring the environment variables.

```
        stage('Ansible Deployment') {
            steps {
                script {
                    sh 'ansible-playbook -i inventory playbook.yml'
                }
            }
        }
    }
```

Jenkins Pipeline Result



Playbook.yml has the permissions to the docker-compose and start docker images.

```yaml
---
- name: Deploy MERN Application
  hosts: all
  vars:
    ansible_python_interpreter: /usr/bin/python3

  tasks:
    - name: Copy Docker Compose file
      copy:
        src: docker-compose.yml
        dest: "docker-compose.yml"

    - name: Run Docker Compose
      command: docker-compose up -d
```

Deployments

Using localhost

npm start: start the front-end

Environment variables for the front-end.

REACT_APP_BASE_URL=http://localhost:5001

nodemon server: Start the front end.

Environment variables for back-end.

jwt_secret=secret
MONGO_URL=mongodb+srv://vivektangudu:viv@cluster0.czt49fi.mongodb.net/lib PORT=5001 NODE_ENV=development

Here we used a cloud database.

Using docker-compose

- While using docker images, we used the Mongo DB docker image to connect to the database.

These images connect between them using a network name “mynetwork”.

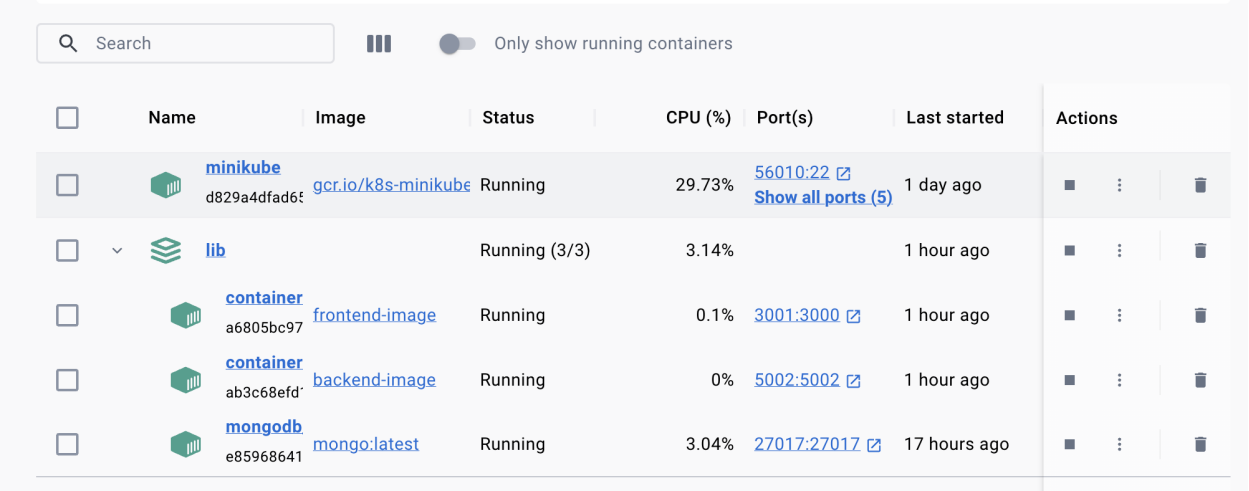
Communication happens using the local host.

- The front-end image exposes the 3000 port and it connects to localhost:3000
- The backend image exposes to 5000 port and it connects to localhost:5000
- Similarly, mongo_db exposes to 27017, and it connects to the backend.

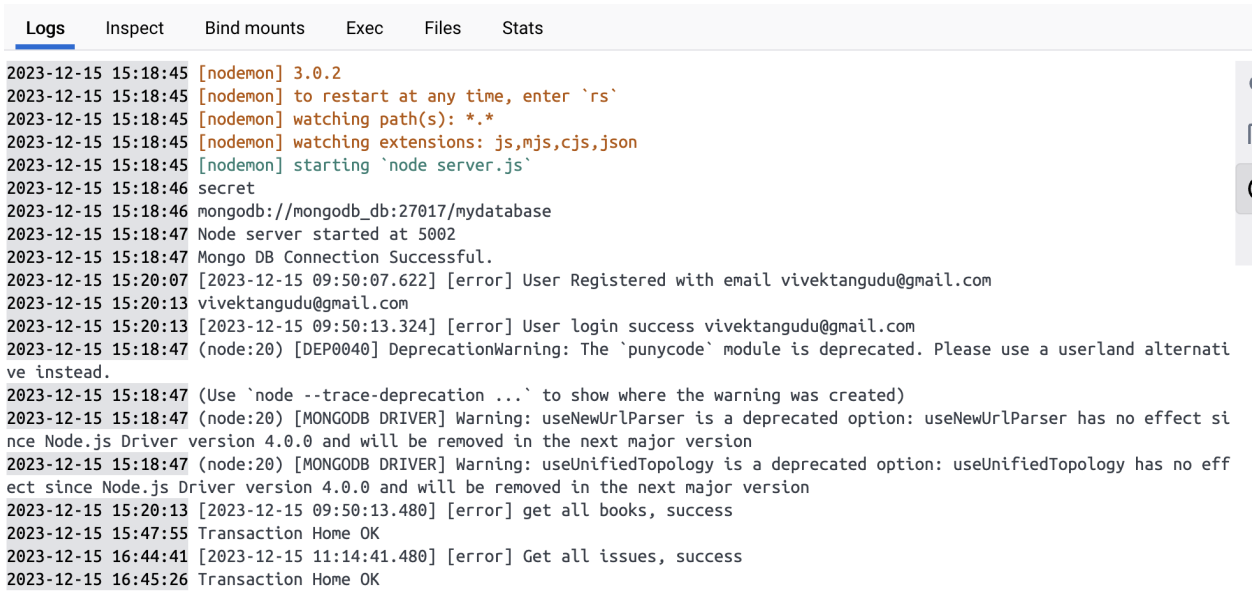
The URL can be **“mongodb://mongodb_db:27027/mydatabase”** or **“mongodb://localhost:27017/mydatabase”**

```yaml
version: '3'
services:
  container_front:
    image: frontend-image
    ports:
      - "3000:3000"
    networks:
      - mynetwork
    depends_on:
      - container_back
    environment:
      - REACT_APP_BASE_URL=http://localhost:5002

  container_back:
    image: backend-image
    ports:
      - "5002:5002"
    networks:
      - mynetwork
    extra_hosts:
      - "host.docker.internal:host-gateway"
    environment:
      - MONGO_URL=mongodb://mongodb_db:27017/mydatabase
      - jwt_secret=secret
      - PORT=5002

  mongodb_db:
    networks:
      - mynetwork
    image: mongo:latest
    ports:
      - "27017:27017"

networks:
  mynetwork:
```

This Docker Compose file defines a multi-container environment for a full-stack application with a frontend, backend, and MongoDB database. It utilizes three services: "container_front" for the frontend, "container_back" for the backend, and "mongodb_db" for the MongoDB database. The frontend and backend containers are connected to the "mynetwork" network, enabling communication. The "container_front" service exposes its application on port 3001, forwarding requests to the backend hosted on port 5002. The backend relies on a MongoDB database with connection details specified in the environment variables. Additionally, the MongoDB container is exposed on port 27017, we can observe the activity in the Mongodb compass.



Deployment from docker-compose.



Logs from the backend docker.

Deployed in render.com

Front-end: A static website. The environment variable is the link of the backend service.

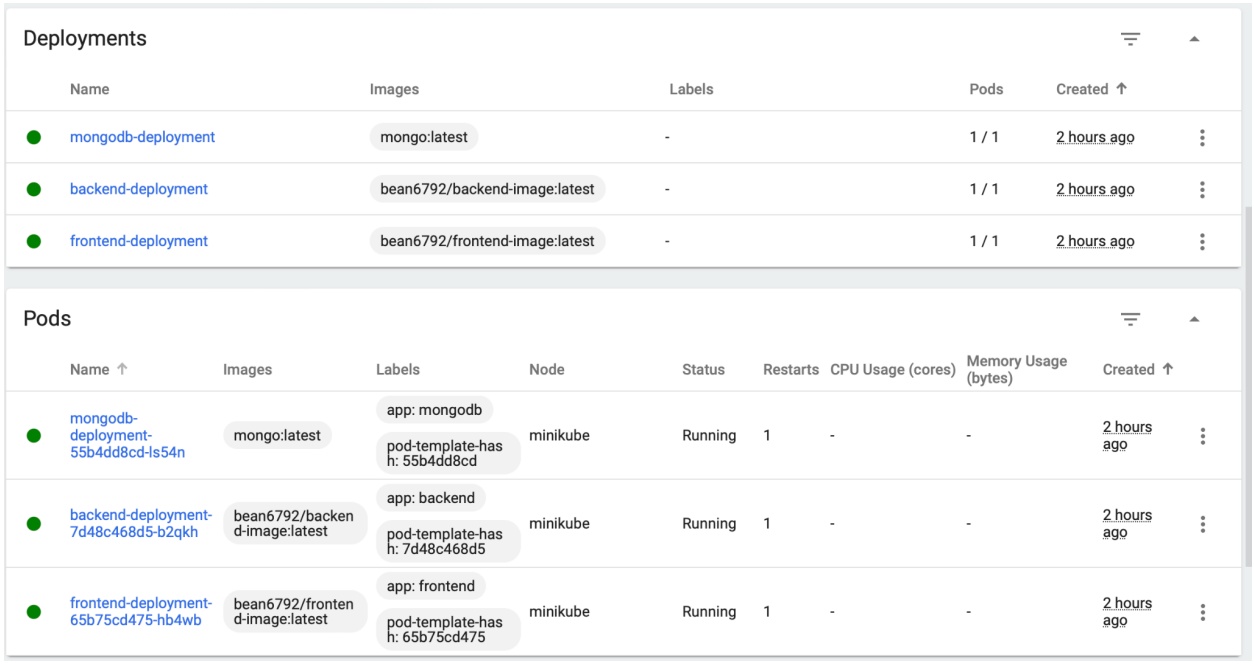
https://library-odfb.onrender.com/

Backend-end: a web service. Mongodb database is the cloud database.

https://back-j9al.onrender.com/

Here the render uses the webhooks from GitHub to build the update the webservices when there is an update in the code.

Deployed in K8



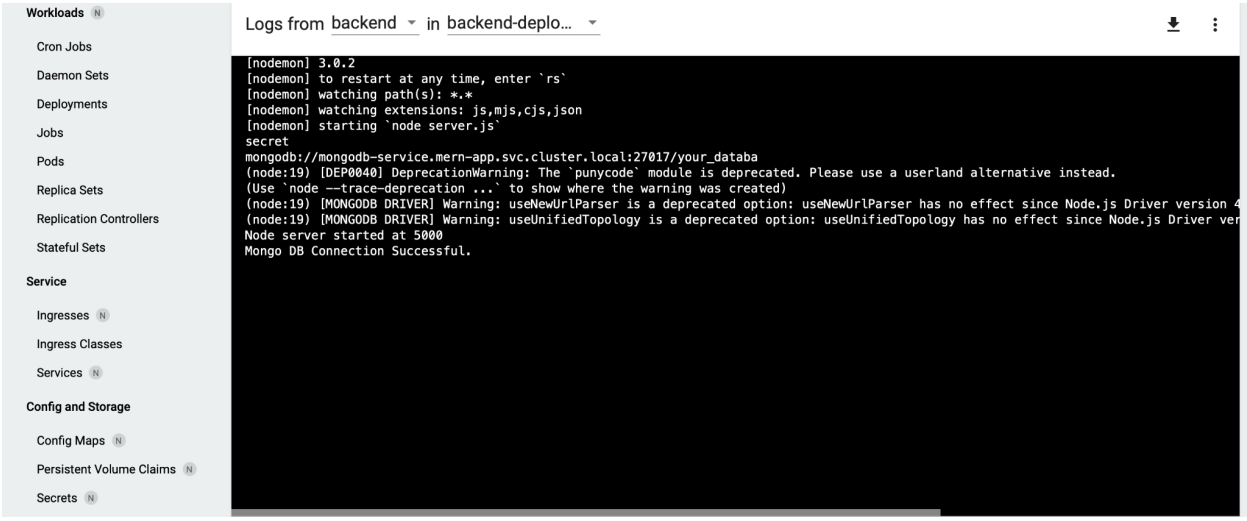
Deployments

Name	Images	Labels	Pods	Created
mongodb-deployment	mongo:latest	-	1 / 1	2 hours ago
backend-deployment	bean6792/backend-image:latest	-	1 / 1	2 hours ago
frontend-deployment	bean6792/frontend-image:latest	-	1 / 1	2 hours ago

Pods

Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created
mongodb-deployment-55b4dd8cd-ls54n	mongo:latest	app: mongodb pod-template-hash: 55b4dd8cd	minikube	Running	1	-	-	2 hours ago
backend-deployment-7d48c468d5-b2qkh	bean6792/backend-image:latest	app: backend pod-template-hash: 7d48c468d5	minikube	Running	1	-	-	2 hours ago
frontend-deployment-65b75cd475-hb4wb	bean6792/frontend-image:latest	app: frontend pod-template-hash: 65b75cd475	minikube	Running	1	-	-	2 hours ago

Deployed in Kubernetes.



These are logs from my backend pod. We deployed the mongo_db image in a pod.

And we use that pod to connect with the backend as the database.

kubectl port-forward <front-end pod name> 3001:3000

kubectl port-forward <backend pod name> 5005:5000

We use the above commands to expose the pods to localhost.

API Documentation

The **"/api/users"** route handles user-related operations, allowing for the management of user data. The **"/api/books"** route facilitates interactions with book-related data, offering endpoints for book-related operations. The **"/api/issues"** route is responsible for handling issues or transactions related to the books, providing endpoints for managing book transactions.

Every response has a tag with success. If the response is true, it means the operation is passed, and if the response is false it means the operation is failed, along with an error message **.**

/api/users/login

Helps to log in to the user with email and password.

```
// login a user
export const LoginUser = async (payload) => {
  try {
    console.log(payload)
    const response = await axiosInstance.post(baseUrl + "/api/users/login", payload);
    console.log(response)
    return response.data;
  } catch (error) {
    throw error;
  }
};
```

There are different errors that can pop up in different situations.

→ If a valid username doesn't exist.

```
// check if user exists
const user = await User.findOne({ email: req.body.email });
console.log(req.body.email)
if (!user) {
  logger.info("User login failed(User does not exist)" + req.body.email);
  return res.send({
    success: false,
    message: "User does not exist",
  });
}
```

→ If a valid password doesn't match.

```
// check if password is correct
const validPassword = await bcrypt.compare(
  req.body.password,
  user.password
);

if (!validPassword) {
  logger.info("User login failed(impor Invalid password) " + req.body.email);
  return res.send({
    success: false,
    message: "Invalid password",
  });
}
```

Passwords are usually encrypted and saved in db. While checking the password, it decrypts the password and checks the password.

→ Last type of message can be a success or it can be a failure.

```
    // create and assign a token
    const token = jwt.sign({ userId: user._id }, process.env.jwt_secret, {
      expiresIn: "1d",
    });
    logger.info("User login success " + req.body.email);
    return res.send({
      success: true,
      message: "Login successful",
      data: token,
    });
  } catch (error) {
    logger.info("User login failed. " + error.message);
    return res.send({
      success: false,
      message: error.message,
    });
  }
});
```

If the login passes, it will send a token to the front end that will be saved in the client system as a cache for further processing.

`api/users/register`

This api works while registering a user. There can be different error messages that can pop up based on the input.

→ Should be a unique email and phone number.

```
router.post("/register", async (req, res) => {
  try {
    // check if user already exists
    const user = await User.findOne({ email: req.body.email });

    if (user) {
      logger.info("User registration failed(email exists) " + req.body.email)
      return res.send({
        success: false,
        message: "Email already exists",
      });
    }

    // hash password
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(req.body.password, salt);
    req.body.password = hashedPassword;

    // create new user
    const newUser = new User(req.body);
    await newUser.save();
    logger.info("User Registered with email " + req.body.email)
    return res.send({
      success: true,
      message: "User created successfully , please login",
    });
  } catch (error) {
    logger.info("User registration Failed " + req.body.email);
    return res.send({
      success: false,
      message: error.message,
    });
  }
});
```

The password is usually saved in db after encrypting the user password.

api/users/get-logged-in-user

We are using cache to store the logged-in user name. If a user re-visits the website. We use the token and verify the user.

```
// get logged in user details
router.get("/get-logged-in-user", authMiddleware, async (req, res) => {
  try {
    const user = await User.findById(req.body.userIdFromToken);
    if (!user) {
      return res.send({
        success: false,
        message: "User does not exist",
      });
    }
    return res.send({
      success: true,
      message: "User details fetched successfully",
      data: user,
    });
  } catch (error) {
    return res.send({
      success: false,
      message: error.message,
    });
  }
});
```

For any operation that is done by the user, a token is sent in the payload to do the operations. It asks like a middleware to authenticate for any operation. The possible errors can be:

```
module.exports = function (req, res, next) {
  try {
    const token = req.headers.authorization;

    if (!token) {
      return res.status(401).json({
        success: false,
        message: "Unauthorized: Token is missing",
      });
    }

    const tokenParts = token.split(" ");
    if (tokenParts.length !== 2 || tokenParts[0] !== "Bearer") {
      return res.status(401).json({
        success: false,
        message: "Unauthorized: Invalid token format",
      });
    }

    const decoded = jwt.verify(tokenParts[1], process.env.jwt_secret);

    if (decoded.userId) {
      req.body.userIdFromToken = decoded.userId;
      next();
    } else {
      return res.status(401).json({
        success: false,
        message: "Unauthorized: Invalid token",
      });
    }
  } catch (error) {
    if (error.name === "TokenExpiredError") {
      return res.status(401).json({
        success: false,
        message: "Unauthorized: Token has expired",
      });
    } else {
      return res.status(500).json({
        success: false,
        message: "Internal server error",
      });
    }
  }
```

api/users/get-all-users

To get all users based on role. This can be used by only admin and staff.

```
// get all users
export const GetAllUsers = async (role) => {
  try {
    const response = await axiosInstance.get(baseUrl + `/api/users/get-all-users/${role}`);
    return response.data;
  } catch (error) {
    throw error;
  }
};
```

api/books/add-book

```
// add a book
router.post("/add-book", authMiddleware, async (req, res) => {
  try {
    console.log(req);
    const newBook = new Book(req.body);
    await newBook.save();
    logger.info("Book added successfully " + req.body.book);
    return res.send({ success: true, message: "Book added successfully" });
  } catch (error) {
    logger.info("Book added failed");
    return res.send({ success: false, message: error.message });
  }
});
```

api/books/update-book/:id

Update book details or copy numbers. We pass the ID of the book and update it accordingly.

```
// update a book
router.put("/update-book/:id", authMiddleware, async (req, res) => {
  try {
    await Book.findByIdAndUpdate(req.params.id, req.body);
    logger.info("Book updated successfully " + req.body.book);
    return res.send({ success: true, message: "Book updated successfully" });
  } catch (error) {
    logger.info("Book update failed");
    return res.send({ success: false, message: error.message });
  }
});
```

api/books/delete-book/:id

Similarly, we can delete books using the id of the book.

api/books/get-all-books

To get a list of all books.

```
// get all books
router.get("/get-all-books", authMiddleware, async (req, res) => {
  try {
    const books = await Book.find().sort({ createdAt: -1 });
    logger.info("get all books, success");
    return res.send({ success: true, data: books });
  } catch (error) {
    logger.info("get all books, failed");
    return res.send({ success: false, message: error.message });
  }
});
```

/api/issues/issue-new-book

When a librarian is issuing a new book. We use this

```
router.post("/issue-new-book", authMiddleware, async (req, res) => {
  try {
    // inventory adjustment (available copies must be decremented by 1)
    await Book.findOneAndUpdate(
      { _id: req.body.book },
      { $inc: { availableCopies: -1 } }
    );

    const newIssue = new Issue(req.body);
    await newIssue.save();
    logger.info("Book issued successfully " + req.body.book)
    return res.send({
      success: true,
      message: "Book issued successfully",
      data: newIssue,
    });
  } catch (error) {
    logger.info("Book issued failed" + req.body.book)
    return res.send({
      success: false,
      message: error.message,
    });
  }
});
```

api/issues/get-user-id-by-email

While issuing a new book, we verify the user's mail address and get the user ID for further things.

```
router.post("/get-user-id-by-email", authMiddleware, async (req, res) => {
  try {
    const email = req.body.email;

    const user = await UserModel.findOne({ email });

    if (user) {
      logger.info("Get id using email " + req.body.email);
      res.send({ success: true, userId: user._id, role: user.role });
    } else {
      logger.info("Get id using email + User not found" + req.body.email);
      res.send({ success: false, message: "User not found" });
    }
  } catch (error) {
    logger.info("Get id using email, Failed " + req.body.email);
    res.send({ success: false, message: error.message });
  }
});
module.exports = router;
```

/api/issues/get-issues

This is primarily used by the admin to get all issues by each user. user details are mentioned in the payload.

```
let response;

if (selectedUser.role === "patron") {
  response = await GetIssues({
    user: selectedUser._id,
  });
} else {
  response = await GetIssues({
    issuedBy: selectedUser._id,
  });
}
```

Similarly, we can delete issues.

`/api/reports/get-reports`

```
export const GetReports = async () => {
  try {
    const response = await axiosInstance.get(baseUrl + "/api/reports/get-reports");
    return response.data;
  } catch (error) {
    console.log(error);
  }
};
```

This api can be used by only the admin. To get all the information in the database.

Testing

We did testing for both the backend and front end using jest and chai.

To do testing we used a separate database in the localhost and tried to add some users and update their details similarly for books.

```
      657c06e89a233bb968d437b3

      at log (routes/booksRoute.js:33:11)


Test Suites: 3 passed, 3 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        1.566 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

These changes are reflected in the db.

Front-end testing, we did for components and the login and register page.

We tried creating different button types and checking for the desired type.

```
build                  package-lock.json      package_123.json       src
vivek@viveks-MacBook-Air client % npm test

> front-side@0.1.0 test
> jest

 PASS  src/__tests__/loader.test.js
 PASS  src/__tests__/button.test.js
 PASS  src/__tests__/user.test.js
 PASS  src/__tests__/register.test.js

Test Suites: 4 passed, 4 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        2.34 s
Ran all test suites.
vivek@viveks-MacBook-Air client %
```

```
process.env.NODE_ENV = 'test';
process.env.MONGO_URL = "mongodb://localhost:27017/testdb"
process.env.PORT = 5003
```

We have changed the environment variables accordingly for testing.

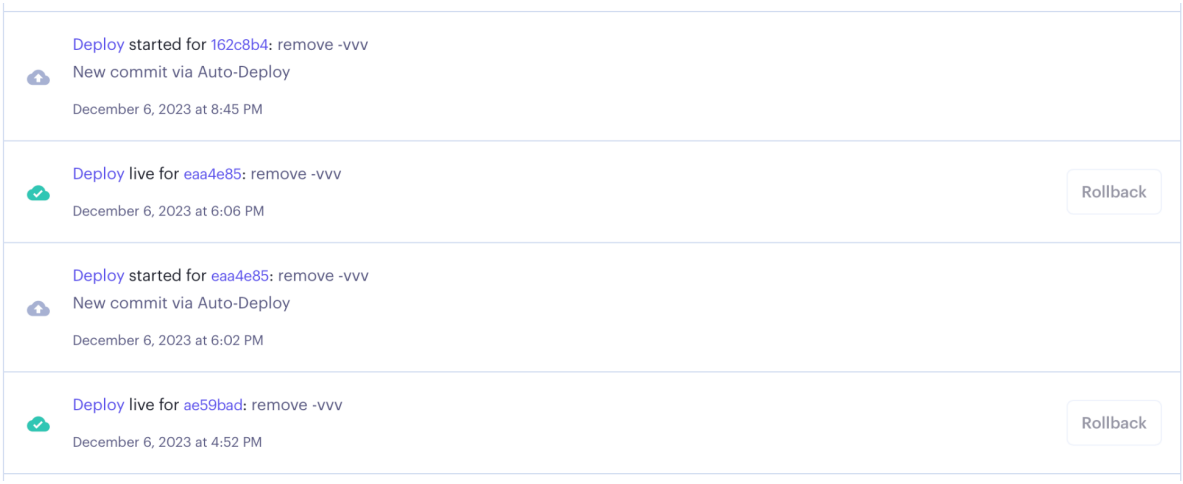
Logging

We used the Winston package for logging. We logged every operation and calls that were made in the backend along with the result, time, and type of deployment (test, info, error).

```
[2023-12-13 17:42:00.513] [info] get all books, success

[2023-12-13 17:49:12.449] [error] get all books, failed
```

Webhooks

Webhooks were used by render.com. As we deployed the frontend and backend in the cloud. We connected the webhook to the cloud to update the build automatically.



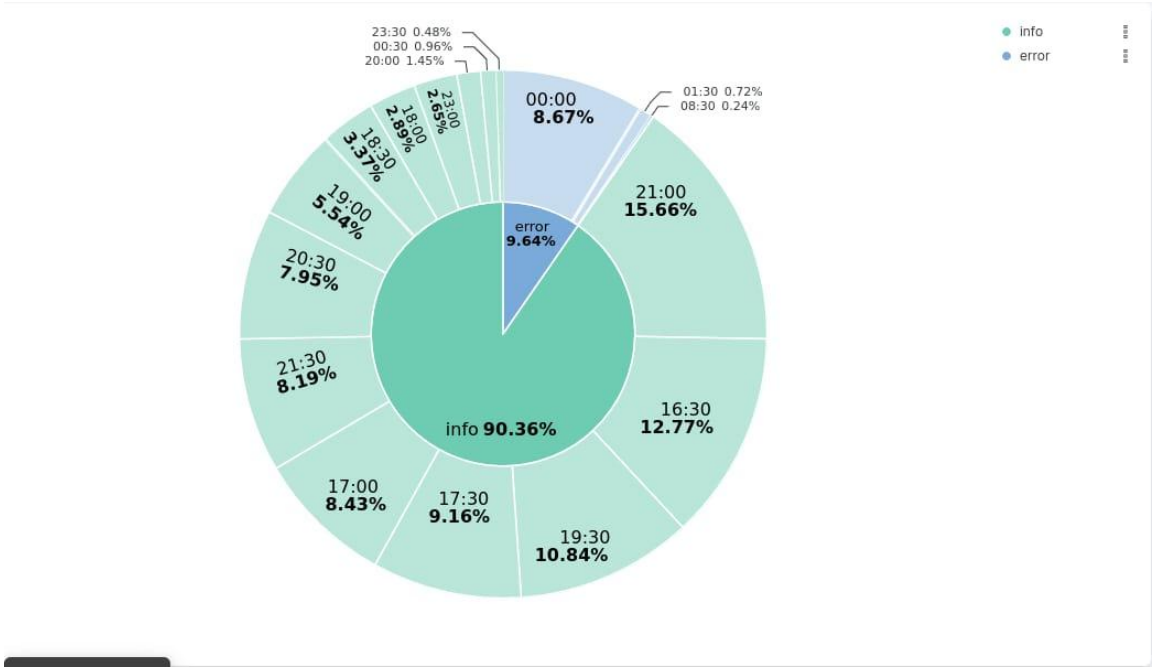
Those were some deployments using webhooks when there was a commit.

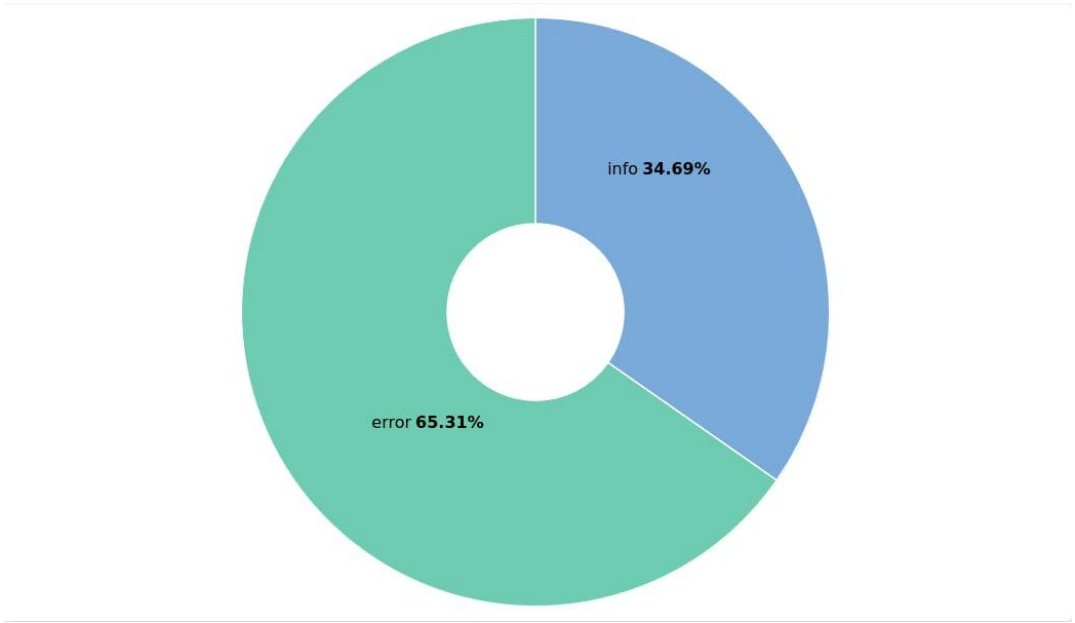
ELK stack

Grok pattern(timestamp (`"YYYY-MM-DD HH:mm:ss.SSS"`) + info/error + message))

ELK stack consists of 3 applications: Elasticsearch, Logstash, and Kibana. ELK stack is useful for collecting logs from all applications, analyzing these logs, and creating visualizations.

Each log has a Timestamp, deployment type (info/ error), and message.





Errors

1) Initially, we had no idea about proxy, later we added an environment variable to api calls. This is not an error but we changed the api calls on every page.
2) Communication between front-end and back-end images after creating docker images. Later we got to know about the network and communication between them.
3) In Harsha's system, process.env.ENVIRONMENT_VARIABLE was not working. But a workaround for this was to specify export in the .env file (which is not required generally) and add an absolute path (not a relative path to the dotenv.config() function.)

```
export jwt_secret=secret
export MONGO_URL=mongodb+srv://vivektangudu:viv@cluster0.czt49fi.mongodb.net/lib
export PORT=5001
export NODE_ENV=development
```

```
require("dotenv").config({
  path: "/home/harsha/Semester-7/SPE/Major/LatestMain/Lib/.env",
});
```

4) In Harsha's system while running Jenkins builds, as Jenkins uses /var/lib/jenkins to install all the npm packages, the root '/' directory ran out of space and the system crashed. After rebooting, the system went into a login loop which prompted the user for username and password repeatedly even after entering the correct credentials. This problem was resolved by booting

into recovery mode and deleting some unnecessary software and older Linux kernels to free up some space in the root directory.

Future Scope

- Replicate the same for renting any other objects like cars or any other things.
- Add functionality for the admin to change the roles of users from the web.
- Instead of giving an image URL for the book, an image upload button can be added.

References

- Class Slides
- stackoverflow.com for debugging various errors and other warnings
- https://tailwindcss.com/docs/ for utility classes
- https://stackoverflow.com/a/11104156/15069364 For process.env error
- Jacob Bhaiya's "Chapter 3 Simple Calculator Program.pdf" for Jenkinsfile
- https://docs.docker.com/get-started/02_our_app/
- https://minikube.sigs.k8s.io/docs/start/ for k8's
- https://docs.docker.com/get-started/docker_cheatsheet.pdf

Note

- @harsha-deep does not have any commits in the Github Repo because @vivektangudu123 did a git force push and all other commits and code was gone.